Hintergrundwissen: Timer

Dienstag, 11. Oktober 2022 12:11

Timer,
Counter u...

heise **Developer**

Timer, Counter und Interrupts

01.08.2016 07:00 Uhr Dr. Michael Stal

In dieser Extraausgabe kommt das Thema Timer zur Sprache. Dabei geht es um mehr als den simplen Aufruf der delay()-Funktion.

In dieser Extraausgabe kommt das Thema Timer zur Sprache. Dabei geht es um mehr als den simplen Aufruf der delay()-Funktion.

Die Funktion *delay()* war bisher unser treuer und ständiger Begleiter in Sachen Zeitmessung. Sie steht standardmäßig in der Arduino-Bibliothek zur Verfügung, und erlaubt millisekundengenaue Wartezeiten. Was will man also mehr? Bei der Entwicklung eingebetteter Systeme – übrigens nicht nur bei Echtzeitfragestellungen – spielt Zeitmessung eine zentrale Rolle. *delay()* ist allerdings ein Mittel fürs Grobe, nicht für ausgefeilte Einsatzgebiete.

Was stört konkret an den simplen Funktionen à la *delay()* oder *millis()*?

- Beschränkte Auflösung: Eine Genauigkeit in Millisekunden mag für Wartezeiten im Maßstab menschlicher Wahrnehmung passabel erscheinen. Im Rahmen technischer Ereignisse ist diese Auflösung ungenügend. Bei einer ATMega-oder ATTiny-CPU auf Arduino-Boards haben wir es in der Regel mit Taktfrequenzen von 8 MHz oder 16 MHz zu tun. Bei 16 MHz dauert ein einzelner Takt 0,0000000625 sec. Das ist im Vergleich zu einer Millisekunde (0,001 sec) ein Faktor von 1:16.000. Während jeder Millisekunde verarbeitet ein ATMega Tausende von Maschinenbefehlen. Daher sind Reaktionszeiten in dieser Größenordnung alles andere als akzeptabel.
- Aktives Warten: Während eines Delays wie *delay(42)* ist der Arduino-Sketch zum aktiven Warten verurteilt. Die gewählte Wartezeit in Millisekunden ist zudem oft Ergebnis einer groben Schätzung nach dem Motto: "Vermutlich braucht die Initialisierung rund 2 Sekunden. Zur Sicherheit verwenden wir aber einfach 3 Sekunden." Das führt zu Ineffizienz.
- Mangelnde Flexibilität: Timernutzung mittels *delay()* dient einzig zum Integrieren von Wartezeiten. Weitere Dienste, insbesondere periodische Trigger wie ein Watchdog-Timer, sind hiermit nicht oder nur unzureichend umsetzbar.

Timer on the Chip

Würde sich ein AVR-Prozessor mit dieser Art von grober Zeitmessung zufrieden geben, müssten wir bei Arduino-Boards viele funktionale Abstriche in Kauf nehmen. So benötigt die Pulse-Width-Modulation an digitalen Ports eine hohe Timerauflösung, um genaue Aktivitätszyklen (Duty-Cycles) einstellen zu können. Benötigen Sie zum Beispiel eine durchschnittliche Ausgangsspannung von 2,765643 V an einem digitalen Ausgang, lässt sich das nur durch Genauigkeiten jenseits der Möglichkeiten von *delay()* oder *millis()* bewerkstelligen.

Bei Bussystemen wie I^2C oder SPI sind Vorgänge im Mikrosekundenbereich typisch und notwendig. Will ein Programm angeschlossene Komponenten über diese Bussysteme ansteuern, wäre ein Auflösung von groben Millisekunden ein Show-Stopper. Nicht zuletzt erweisen sich für die Kontrolle von Servomotoren oder zur Tonerzeugung zeitliche Bedingungen als notwendig, die durch Millisekunden-getriggerte Timer schlicht nicht umsetzbar wären.

Da wir aus bisheriger Erfahrung wissen, dass Arduino-Boards sehr gute Unterstützung für PWM, Bussysteme, Servo-Ansteuerung und Tonerzeugung leisten, stellen sich zwei Fragen:

1. Wie schafft es ein AVR-Prozessor intern, die dafür erforderlichen zeitlichen Auflösungen umzusetzen?
2. Können wir uns als Entwickler diese Funktionen ebenfalls zunutze machen?

Den Takt angeben

Für die genannten Aufgaben integrieren die AVR-Microcontroller diverse Timer mit zugeordneten Zählregistern von 8 oder 16 Bits Breite. Diese Register starten mit einem initialen Wert von 0. Ihr Inkrementieren, also das eigentliche Hochzählen, erfolgt automatisch und periodisch. Laufen die jeweiligen Register über, wird ein Timer-Überlauf-Interrupt ausgelöst.

Ein häufiger Irrtum lautet übrigens, dass die CPU des Arduino die Timer antreibt. Die Timer eines Arduino bzw. eines AVR-Mikrocontrollers von ATmel sind von der CPU bzw. MCU unabhängig. Diese Tatsache sollten Sie sich bei den nachfolgenden Diskussionen vor Augen halten.

Um eine möglichst hohe Auflösung zu erhalten, könnte der erste Ansatz darin liegen, die Updates der Counter (d.h. deren Inkrementieren) synchron zum Prozessortakt vorzunehmen. Das hat allerdings einen entscheidenden Schönheitsfehler. Bei einer angenommenen Taktfrequenz von 16 MHz wäre der Überlauf eines 8-Bit-Timers nach 16 Mikrosekunden erreicht, der eines 16-Bit-Registers nach rund 4,1 Millisekunden.

Das ist natürlich vorteilhaft für kurzzeitige Zeitintervalle. Was aber, wenn wir längere Zeiträume überdecken wollen? Um dies zu ermöglichen, bieten die Mikrocontroller sogenannte Prescaler. Diese konfigurieren, nach wie vielen Taktzyklen das System ein Zählregister inkrementieren soll. Mögliche Werte liegen bei 8, 64, 256 oder 1024. Eine Prescale-Einstellung von 1024 führt beispielsweise zum Inkrementieren des Zählers jeweils nach 64 Mikrosekunden bzw. 1024 Taktzyklen bei 16 MHz Taktfrequenz, sodass ein 16-Bit-Zähler erst nach 4,2 Sekunden überläuft.

Um präzise Intervalle zu programmieren, lassen sich Timer-Register mit Zählerständen vorbelegen statt sie bei 0 starten zu lassen. Einmal angekommen, wir würden gerne alle 0,5 Sekunden eine LED abwechselnd ein und ausschalten. Die gewünschte Frequenz des Timers wäre somit 2 Hz. Ein Takt besteht im Auslösen eines Timer-Überlaufs. Wie genau lässt sich dies erreichen?

Wir haben es mit folgenden Parametern zu tun:

- *bits* definiert die Größe des Zählerregister in Bits, etwa 16 für einen 16-Bit-Timer.
- *maxcount* entspricht dem maximalen Zahlenwerts 2^{bits}.
- *prescale* ist der oben erläuterte konfigurierbare Prescalewert, also die Zahl der Taktzyklen bis ein weiteres Inkrementieren des Timeregisters erfolgt.
- *cpufreq* repräsentiert die CPU-Frequenz. Der Taktzyklus berechnet sich folglich aus *1 / cpufreq*.
- *initcount* ist der vorbelegte Startwert des Zählregisters.
- *count* ist die notwendige Zahl von Inkrementierungen, um einem Timeroverflow auszulösen. Es gilt: *count = maxcount - initcount*.
- *deltaT* bezeichnet das gewünschte Zeitintervall bis zum Auslösen des Timer-Overflows. Man könnte auch definieren: *deltaT = 1 / timerfreq* (gewünschte Zahl von Timer Overflows pro Sekunde).

Es gilt *prescale / cpufreq * count = deltaT*

=> *count = deltaT * cpufreq / prescale*

=> *maxcount - initcount = deltaT * cpufreq / prescale*

=> *initcount = maxcount - deltaT * cpufreq / prescale*

Beispielsrechnung: Alle 0,5 Sekunden soll ein Timer-Overflow-Interrupt stattfinden.

- Wir verwenden einen 16-Bit-Timer: *bits* = 16 => *maxcount* = 2^{16} = 65536.
- Wir benötigen einen Timer Overflow pro halbe Sekunde. *deltaT* = 0,5 sec = *1 / timerfreq*
- Die Taktfrequenz des Arduino-Board beträgt *cpufreq* = 16 MHz = 16.000.000 Hz
- Als Prescale-Wert liegt *prescale* = 256 vor.

Der Timer startet statt mit 0 mit folgendem Anfangszählerstand *initcount* = 65.536 - 8.000.000/256 = 34.286

Das Timer-Register muss initial mit 34.286 starten, damit bis zum Timer Overflow – bei Überschreiten von 65.636 – genau eine halbe Sekunde vergeht. In jedem Durchlauf der Interrrupt-Service-Routine ist der Zähler jeweils wieder mit 34.286 initialisieren.

Ein entsprechender Sketch könnte wie folgt aussehen. Auf die darin erwähnten Register kommen wir später noch zu sprechen.

```
#define ledPin 13

void setup()
{
  pinMode(ledPin, OUTPUT);  // Ausgabe LED festlegen
```

```
  // Timer 1
  noInterrupts();           // Alle Interrupts temporär abschalten
  TCCR1A = 0;
  TCCR1B = 0;

  TCNT1 = 34286;            // Timer nach obiger Rechnung vorbelegen
  TCCR1B |= (1 << CS12);    // 256 als Prescale-Wert spezifizieren
  TIMSK1 |= (1 << TOIE1);   // Timer Overflow Interrupt aktivieren
  interrupts();             // alle Interrupts scharf schalten
}
// Hier kommt die selbstdefinierte Interruptbehandlungsroutine
// für den Timer Overflow
ISR(TIMER1_OVF_vect)
{
  TCNT1 = 34286;             // Zähler erneut vorbelegen
  digitalWrite(ledPin, digitalRead(ledPin) ^ 1); // LED ein und aus
}

void loop()
{
  // Wir könnten hier zusätzlichen Code integrieren
}
```

World of Timers

Ein Arduino weist nicht nur einen einzelnen Timer sondern mehrere Timer auf. Kein Wunder, sind Timer doch essenzielle Grundkomponenten für verschiedene Aufgaben eines Mikrocontrollers.

- Timer 0 (8 Bit) Verwendet für Funktionen wie *delay()*, *millis()*, *micros()*
- Timer 1 (16 Bit) Verwendet von der Servo-Bibliothek
- Timer 2 (8 Bit) Verwendet von der Tone-Bibliothek
- Timer 3 (16 Bit) Nur Mega
- Timer 4 (16 Bit) Nur Mega
- Timer 5 (16 Bit) Nur Mega

Beim Arduino:

- PWM Pins 5 und 6 kontrolliert durch Timer 0
- PWM Pins 9 und 10 kontrolliert durch Timer 1
- PWM Pins 3 und 11 kontrolliert durch Timer 2

Beim Arduino Mega:

- PWM Pins 4 und 13 kontrolliert durch Timer 0
- PWM Pins 11 und 12 kontrolliert durch Timer 1
- PWM Pins 9 und 10 kontrolliert durch Timer 2
- PWM Pins 2, 3 und 5 kontrolliert durch Timer 3
- PWM Pins 6, 7 und 8 kontrolliert durch Timer 4
- PWM Pins 44, 45 und 45 kontrolliert durch Timer 5

Es gibt zusätzlich diverse Einschränkungen zu beachten:

- Pin 11 des Arduino ist zugleich Pin mit PWM-Fähigkeit und Master-Out-Slave-In-Pin des SPI-Busses. Demzufolge lassen sich beide Funktionen nicht gleichzeitig nutzen.
- Für die Tonerzeugung ist mindestens Timer 2 im Einsatz. Daher lassen sich die Pins 3, 11 (Arduino) bzw. 9, 10 (Arduino Mega) nicht für PWM nutzen, solange die Funktion *tone()* im Einsatz ist.
- Beim Anschluss von Servos müssen sich Timer exklusiv dieser Aufgabe widmen, weshalb sich die Zahl der Digitalpins mit PMW-Unterstützung reduziert.

Timer-Register

Wie im Sketch weiter oben ersichtlich, erfolgt die Steuerung der Timer-Funktionalität über verschiedene Register. Das Symbol µ repräsentiert die Nummer des jeweiligen Timers, also 0, 1, 2, ..., . . TCNTµ ist daher das Zählregister von Timer µ. Das Zählregister für Timer 1 lautet dementsprechend TCNT1, das für Timer 0 TCNT0.

Der Einfachheit halber beziehen sich die nachfolgenden Diskussionen wie auch die beiden Beispiel-Sketches auf Timer 1. Des Weiteren erspare ich Ihnen eine Aufzählung sämtlicher Details, sondern fokussiere mich auf die relevanten Eigenschaften.

TCCR1A (Timer Counter/Control Register): die Flags PWM10 und PWM11 erlauben eine Festlegung der Auflösung für den Fall, dass Timer 1 zur PWM-Steuerung dient. Ausgangsbasis sei die Vereinbarung *TCCR1A = 0;*:

- Kein PWM: `no-op`
- 8-Bit PWM: `TCCR1A |= (1 << PWM10);`
- 9-Bit PWM: `TCCR1A |= (1 << PWM11);`
- 10-Bit PWM: `TCCR1A |= (1 << PWM 10); TCCR1A |= (1 << PWM11);`

TCCR1B (Timer Counter/Control Register): Konfiguration des Prescaler.

- Kein Prescaler: `TCCR1B = 0; TCCR1B |= (1 << CS10);`
- Prescale = 8: `TCCR1B = 0; TCCR1B |= (1 << CS11);`
- Prescale = 64: `TCCR1B = 0; TCCR1B |= (1 << CS10); TCCR1B |= (1 << CS11);`
- Prescale = 256: `TCCR1B = 0; TCCR1B |= (1 << CS12);`
- Prescale = 1024: `TCCR1B = 0; TCCR1B |= (1 << CS10); TCCR1B |= (1 << CS12);`

Weitere Kombinationen ermöglichen die externe Steuerung über den T1-Pin.

TCNT1 (Timer/Counter Register): d.h. der eigentliche Zähler.

OCR1 (Output Compare Register): Ist der Zähler in TCNT1 gleich dem Inhalt des OCR1, erfolgt ein Timer Compare Interrupt.

ICR1 (Input Capture Register, nur für 16-Bit-Register): Messung der Zeit zwischen zwei Flanken des Input Capture Pins, die durch externe Schaltungen zustande kommen. Lässt sich auch zur Messung der Umdrehungszahl eines Motors einsetzen. Wird auch über Einstellungen von TCCR1A mit beeinflusst.

TIMSK1 (Timer/Counter Interrupt Mask Register): hier lassen sich Timer Interrupts unterbinden oder erlauben.

- Scharf schalten des Output Compare Interrupts: `TIMSK1 |= (1 << OCIE1A)`
- Scharf schalten des Timer Overflow Interrupts (16 Bit): `TIMSK1 |= (1 << TOIE1)`
- Scharf schalten des Timer Overflow Interrupts (16 Bit): `TIMSK1 |= (1 << TOIE0)`

TIFR1 (Timer/Counter Interrupt Flag Register): Hier lassen sich noch unverarbeitete Interrupts feststellen. Die Bits korrespondieren mit denen von TIMSK1.

Alternative Methode CTC

Statt einen Interrupt bei Überlauf eines Timer-Registers auszulösen wie im oberen Sketch, gibt es es die alternative Option namens CTC (Clear Timer on Compare Match). Bei dieser vergleicht der Mikrocontroller, ob der Inhalt des Zählerregisters identisch mit dem Inhalt des zum Timer gehörigen OCR (Output Compare Registers) ist. Falls ja, wird ein Timer Compare Interrupt ausgelöst und das Register auf 0 zurückgesetzt. Wiederum soll jede halbe Sekunde ein Interrupt stattfinden.

Bei einem Prescaling von 256 und einer Taktfrequenz von 16 MHz können wir die obige Formel für count anwenden: *count = deltaT * cpufreq / prescale = 0.5 * 16.000.000 / 256 = 31.256.*

Obiger Sketch würde sich in diesem Fall also ändern in:

```
#define ledPin 13

void setup()
{
  pinMode(ledPin, OUTPUT);  // Ausgabe LED festlegen

  // Timer 1
  noInterrupts();           // Alle Interrupts temporär abschalten
```

```
  TCCR1A = 0;
  TCCR1B = 0;
  TCNT1 = 0;                // Register mit 0 initialisieren
  OCR1A = 31250;            // Output Compare Register vorbelegen
  TCCR1B |= (1 << CS12);    // 256 als Prescale-Wert spezifizieren
  TIMSK1 |= (1 << OCIE1A);  // Timer Compare Interrupt aktivieren
  interrupts();             // alle Interrupts scharf schalten
}
// Hier kommt die selbstdefinierte Interruptbehandlungsroutine
// für den Timer Compare Interrupt
ISR(TIMER1_COMPA_vect)
{
  TCNT1 = 0;                // Register mit 0 initialisieren
  digitalWrite(ledPin, digitalRead(ledPin) ^ 1); // LED ein und aus
}

void loop()
{
  // Wir könnten hier zusätzlichen Code integrieren
}
```

Zusammenfassung

In diesem Extra ging es um den Umgang mit ausgefeilten Timer-Funktionen bei Prozessoren der ATmel-Familie (ATMega, ATTiny). Damit sollten Sie jetzt ein tiefergehendes Verständnis der Thematik besitzen. Wollen Sie es noch detaillierter wissen, verweise ich Sie auf Dokumente des Herstellers ATmel wie zum Beispiel **das hier** [1].

Mehr über die praktische Anwendung dieser Funktionalität erfahren Sie in fortgeschrittenen Anwendungen wie etwa **Sound Synthesis** [2], **PWM** [3] oder **Servomotoren** [4]. Eine weitere lohnenswerte Lektüre ist dieser **Artikel von Adafruit.** [5]

Es ist aber auch eine gute Idee, das Gelernte durch eigene Experimente zu vertiefen.

MEHR INFOS

Was bisher geschah:

- Arduino goes ESP8266 [6]
- Bright Side of Arduino [7]
- Genuino bzw. Arduino unplugged [8]
- Arduino-/IoT-Extra – Bibliotheken selbst implementieren [9]

[13] https://www.heise.de/blog/Schritt-fuer-Schritt-3227165.html
[14] https://www.heise.de/blog/Anwendungen-mit-Servo-Motoren-3226402.html
[15] https://www.heise.de/blog/Gut-motorisiert-3224170.html
[16] https://www.heise.de/blog/Arduino-fuer-Fledermaeuse-3221620.html
[17] https://www.heise.de/blog/Auf-Distanz-3221542.html
[18] https://www.heise.de/blog/Anschluss-von-LCD-Displays-3217991.html
[19] https://www.heise.de/blog/Lauschen-mit-Sensoren-3217195.html
[20] https://www.heise.de/blog/JavaScript-an-Arduino-Bitte-leuchten-3212913.html
[21] https://www.heise.de/blog/Beispielsanwendung-3212826.html
[22] https://www.heise.de/blog/DiY-Arduino-on-a-Breadboard-3198567.html
[23] https://www.heise.de/blog/Crashkurs-Elektronik-fuer-IoT-Anwendungen-Teil-3-von-3-3198491.html
[24] https://www.heise.de/blog/Crashkurs-Elektrizitaet-Teil-2-von-3-3197750.html
[25] https://www.heise.de/blog/Crashkurs-Elektrizitaet-3196453.html
[26] mailto:michael.stal@gmail.com

- Kleiner, tiefer, kürzer ... Entwicklung einer Wetterstation [10]
- RESTful mit CoAP [11]
- Kommunikation über MQTT [12]
- Schritt für Schritt [13]
- Anwendungen mit Servo-Motoren [14]
- Gut motorisiert mit Gleichstrommotoren [15]
- Arduino für Fledermäuse [16]
- Bewegungserkennung durch Infrarot-Strahlung [17]
- Anschluss von LCD-Displays über den IIC-Bus [18]
- Lauschen mit Sensoren [19]
- JavaScript an Arduino: Es werde Licht [20]
- Anschluss vieler LEDs an wenige Ausgänge [21]
- Do-it-Yourself-Projekt: iXduino – Arduino on a Breadboard [22]
- Crashkurs "Elektronik für IoT-Anwendungen" (Teil 3 von 3) [23]
- Crashkurs "Elektronik für IoT-Anwendungen" (Teil 2 von 3) [24]
- Crashkurs "Elektronik für IoT-Anwendungen" (Teil 1 von 3) [25]

([26])

URL dieses Artikels:

https://www.heise.de/-3273309

Links in diesem Artikel:

[1] http://www.atmel.com/Images/Atmel-2505-Setup-and-Use-of-AVR-Timers_ApplicationNote_AVR130.pdf
[2] http://makezine.com/projects/make-35/advanced-arduino-sound-synthesis/
[3] https://arduino-info.wikispaces.com/Arduino-PWM-Frequency
[4] https://www.arduino.cc/en/Reference/Servo
[5] https://learn.adafruit.com/multi-tasking-the-arduino-part-2/timers
[6] https://www.heise.de/blog/Arduino-goes-ESP8266-3240085.html
[7] https://www.heise.de/blog/Bright-Side-of-Life-3269150.html
[8] https://www.heise.de/blog/IoT-mit-dem-Genuino-Arduino-MKR1000-3235840.html
[9] https://www.heise.de/blog/Extrablatt-Bibliotheken-selbst-implementieren-3266419.html
[10] https://www.heise.de/blog/Kleiner-tiefer-kuerzer-3262799.html
[11] https://www.heise.de/blog/RESTful-mit-CoAP-3251225.html
[12] https://www.heise.de/blog/Kommunikation-ueber-das-Ethernet-Shield-mit-MQTT-3238975.html